A Report Generator for Database and Web Applications

Woei-Kae CHEN\(^{(a)}\), Member and Pin-Ying TU\(^{(b)}\), Nonmember

SUMMARY Report generation is one of the most important tasks for database and e-commerce applications. Current report tools typically provide a set of predefined components that are used to specify report layout and format. However, available layout options are limited, and WYSIWYG formatting is not allowed. This paper proposes a four-phase report generation process to overcome these problems. The first phase retrieves source tables from the database. The second phase reorganizes the layout of the source tables by transferring the source tables into a set of new flat tables (in the first normal form). The third phase restructures the flat tables into a nested table (report) by specifying the report structure. The last phase formats the report with a WYSIWYG format editor supporting a number of formatting rules designed specifically for nested reports. Each phase of the proposed process supports visual programming, giving an easy-to-use user interface and allowing very flexible report layouts and formats. A visual end-user-programming tool, called TPS, is developed to demonstrate the proposed process and show that reports with sophisticated layouts can be created without writing low-level report generation programs.

key words: report generation, visual programming, layout transformation, report structure, and report formatting

1. Introduction

Web 2.0, the second stage of web evolution, is attracting the attention of IT professionals, business, and web users [1]. Since dynamic web pages consisting of database reports are essential to almost every transaction in a web application, report generation is arguably the most important task for database and web applications [26]. As an example, Fig. 1 is a report displaying the seat arrangement of the shows in a theater. The report displays the customers who made the reservations and the price for each row. To improve readability, each show is presented in a distinct table. In fact, Show A, B, C, and D are subtables of a larger table (indicated by the dashed lines). In this paper, a report with tables consisting of subtables is called a nested report.

Report generation is the process of converting source tables to a report. Two source tables (Fig. 2) retrieved from a database are used to build Fig. 1. The Reservations table records seat reservations. For example, the seat of Mr. A is reserved at row 1 and column 4. The Prices table contains the price of the seat. For example, a seat in row 1 costs 10 dollars for Show A. Since the layout of the source tables differs from that of the report, a layout transformation is needed. In this case, the reservations (from Reservations table) for Show A are rearranged into a two-dimensional format (Show A subtable of Fig. 1). In addition, the price (from Prices table) information is copied to the corresponding row of the show. Then, formatting operations such as arranging text and color styles are applied to the report (e.g., set the background color of the price column gray).

In general, a report is designed by a report designer, who is familiar with the application domain and can define the report layout so that the report is easy to understand. A report designer is not necessarily a programmer. Therefore, researches [2]–[4] as well as commercial [5]–[8] and open-source [9] tools have been proposed and developed to simplify report generation. Most tools [6]–[9] offer a report layout template editor that uses a set of predefined components (e.g., grid, list, matrix, etc.) to specify layouts. However, this approach can only support a limited number of layouts. If the target layout is not supported, a dedicated program must be written to produce the report. Though writing such a program for a report like Fig. 1 is not difficult for a professional programmer, it is generally not suitable for report designers. Another problem is formatting. The report layout template editor can be used to specify report formats. However, WYSIWYG (What You See Is What You Get) style of formatting is not supported, because the format is specified on the report template, not on the final report. Thus, a better approach is desired.

To overcome the above problems, this paper focuses on
end-user programming for report generation. A four-phase report generation process (Fig. 3) is proposed. The first phase, data query, retrieves a set of source tables, $D_{source}$, from a database. Since the layout of $D_{source}$ may not be suitable for direct presentation, the second and third phases handle layout transformation. The second phase, tabular transformation, transforms the layout of $D_{source}$ into a new layout $D_{flat}$ in the first normal form (flat tables). A set of table operations is proposed to perform the transformation. After the second phase, the display order of $D_{flat}$ is undefined and, sometimes, flat tables need to be converted into nested ones (e.g., Fig. 1). Thus, the third phase, nested structure, restructures the display order and composition of $D_{flat}$. A series of tree nodes is used to define a report structure tree that specifies the structure of the report, produces nested results, and offers aggregation functions. The last phase, report format, formats the final report for the target display environment (e.g., PDF or HTML) by using a WYSIWYG editor with a number of formatting rules designed specifically for nested reports.

The earlier work of this research addressed the techniques used in the second [25] and the third phase [26]. This paper unifies the earlier work and a new report formatting methodology to offer a complete visual report design tool, called Table Presentation System (TPS). With the aid of TPS, the proposed process (Fig. 3) is easy to follow for a report designer without programming expertise. Thus, TPS is suitable for end-user programming. Note that many researches and tools have been proposed for visual query (e.g., [14], [15]), which can be used to accomplish the first phase. Therefore, this paper does not discuss the data query phase.

The rest of this paper is organized as follows. Section 2 describes the visual report tool, TPS. Section 3 explains the tabular transformation phase. Section 4 describes the nested structure phase. Section 5 explains the report format phase. Section 6 describes related work. Section 7 gives an evaluation of the proposed approach. A comparison of the proposed approach with the other approaches is given in Sect. 8. A conclusion is given in Sect. 9.

2. TPS: A Visual Report Development Tool

Before continuing to the explanation for each phase of the proposed process, we will explain the proposed visual report-development tool, called TPS (Table Presentation System), first. The explanation of TPS facilitates the introduction of the concepts described in Sects. 3, 4, and 5. TPS is implemented in Java with JDBC as its database interface and supports both client/server and web-based applications (configured as a dynamic web-page generator). Figure 4 illustrates how TPS is used. TPS is largely based on programming by example methodology [4]. When a report is designed, the report designer uses example data to create an example report, and during the creation of the report, TPS recognizes report designer’s actions and automatically generates the script needed to produce the report. When a report is designed, the report designer follows the proposed report-generation process (Fig. 3) and uses the development GUI to create the report (the arrow labeled (3)). The development GUI invokes an interpreter to perform the report designer’s operations (14) and, if necessary, obtains source tables (15) from the database. Thus, the report designer can directly view the resulting report (i.e., examples). All operations are stored (16) into an XML script file that can be used to reproduce the report.

When a report is displayed, a report viewer acquires the report (a dynamic web page) from a web server (17). The server delegates the request to the interpreter (18), which loads (19) and executes the specified script file. The information contained in the script file directs the interpreter to obtain source tables from the databases (20), transform layouts, and perform formatting to reproduce the report. The report is delivered to the server (21), and then to the report viewer in either HTML or PDF format (22).

Figure 5 shows TPS development GUI. There are three parts. Part A lists tables obtained directly from database ($D_{source}$). Part B lists the tables produced by table operations ($D_{flat}$). Part C consists of five selectable interfaces, namely Config, Input, Table Operations, Report Structure, and Formatting, following the ordering of the proposed process (Fig. 3). A report designer uses one interface at a time (in the above order) to create reports. The first two interfaces (Config and Input) provide database and SQL interfaces that
are used to retrieve $D_{source}$. Note that it is possible to employ a visual query methodology for input interface. However, as explained earlier, it is not the focus of this paper. Therefore, a simple SQL interface is used instead. The rest of the interfaces (Table Operations, Report Structure, and Formatting) are used to perform layout transformation and formatting. We will address them in the following sections.

3. Tabular Transformation

The goal of the second phase, tabular transformation, is to transfer the layout of source tables into a new layout that is suitable for presentation. A set of table operations is proposed to enable visual programming in this phase. The idea is to restrict the output of this phase, $D_{flat}$, into tabular formats in the first normal form. Since $D_{source}$ is inherently tabular, by restricting $D_{flat}$ also into tabular formats, the layout transformation problem is reduced to the transformation of data cells between tables. A single table operation transfers (alters) the layout of a table in a small way. By cascading a sequence of table operations, $D_{source}$ can be arbitrarily transformed to the desired $D_{flat}$. For example, to generate a report like Fig. 1, the two source tables shown in Fig. 2 must be converted to the table shown in Fig. 6. We will briefly discuss the proposed table operations and explain how to accomplish the conversion using these operations. For a more complete and formal mathematical definition of the operations, please refer to the “formal definition of table operations” document given in [27].

Table operations are categorized into primitive and derived operations. A primitive operation is an operation that cannot be decomposed. A derived operation, on the other hand, is a composition of a sequence of primitive operations. This section describes the mathematical concept of each table operation. For convenience, a table is represented as a matrix with homogeneous elements. For example, an $m \times n$ table $T$ is

$$T = \begin{bmatrix} t_{11} & \ldots & t_{1j} & \ldots & t_{1n} \\ \vdots & \ddots & \vdots \\ t_{n1} & \ldots & t_{nj} & \ldots & t_{nn} \end{bmatrix},$$

(1)

where $t_{ij}$ is the element at the $i^{th}$ row and $j^{th}$ column. As a convention, the letters $A$ and $B$ are used to denote source tables; $R$ is used to denote a result table; lower case letters are used to denote the elements in a table (e.g., $a_{ij}$, $b_{ij}$, and $r_{ij}$ are the elements in $A$, $B$, and $R$, respectively); and $T_m$ and $T_n$ denote the number of rows and columns of $T$, respectively. In TPS, the source tables can be obtained either from databases, XML files, or the report designer’s inputs.

3.1 Primitive Operations

There are five primitive operations: Select, Contract, Expand, Merge, and Match. These operations can be used to derive other operations. A Select operation filters a subset of elements out of a source table. Given a list of row indices $\langle a_1, a_2, \ldots, a_m \rangle$ and a list of column indices $\langle \beta_1, \beta_2, \ldots, \beta_n \rangle$ to be selected from a table, the Select operation is defined as:

$$R = \text{Select}_{rows} = \langle a_1, a_2, \ldots, a_m \rangle, \text{columns} = \langle \beta_1, \beta_2, \ldots, \beta_n \rangle \ (A),$$

(2)

where $R$ has $m \times n$ elements defined by $r_{ij} = a_{ij} \beta_j$. For example:

$$\text{Select}_{rows} = (1, 3), \text{columns} = (2, 4) = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix},$$

(3)

The lists of row and column indices can be expressed as either $(l_1, l_2, \ldots, l_q)$ to denote a list of target rows or columns, $\langle * \rangle$ to denote all rows or all columns, or $\langle \text{start}, \text{end}; \text{step} \rangle$ to denote $(\text{start}, \text{start} + \text{step} + \ldots, \text{end})$.

Given two tables $A$ and $B$ and two positive integers $dx$ and $dy$, a Merge operation transfers the location of $B$ to $(dx + 1, dy + 1)$ and merges $B$ with $A$. The Merge operation is defined as:

$$R = \text{Merge}_{x = dx, y = dy} (A, B),$$

(4)

where $R$ has max$(A_m, B_n + dy) \times$ max$(A_n, B_m + dx)$ elements defined by $r_{ij} = a_{ij} \cup b_{ij}$, where $\alpha$ and $\beta$ are

$$\alpha = \begin{cases} a_{ij} & \text{if } 1 \leq i \leq A_m \text{ and } 1 \leq j \leq A_n \\ \emptyset & \text{else} \end{cases} \quad (5)$$

$$\beta = \begin{cases} b_{ij} & \text{if } 1 \leq i - dy \leq B_m \text{ and } 1 \leq j - dx \leq B_n \\ \emptyset & \text{else} \end{cases} \quad (6)$$

It is possible that after a Merge operation, the resulting table contains empty elements or elements with composite values. For example:

$$\text{Merge}_{dx = 1, dy = 1} = \begin{bmatrix} x_{11} & x_{12} & y_{11} & y_{12} \\ x_{21} & x_{22} & \emptyset & \emptyset \\ \emptyset & \emptyset & y_{21} & y_{22} \end{bmatrix},$$

(7)

Note that “,” is used to denote an element with multiple values.

A Match operation can perform either 1D or 2D matching of two tables. A 1D matching is similar to a database join operation and a 2D matching is a 2-dimensional version of the matching. A Match operation is defined as:

$$R = \text{Match}_{dimension = d} (A, B),$$

(8)

where $d$ is either 1 or 2. For 2D matching, $R$ has $(A_m - 1) \times (A_n - 1)$ elements defined by $r_{ij} = \{b_{i+1} : 1 \leq x \leq B_m \text{ and } b_{i+1} = a_{(i+1)j} \}$ and $b_{i+1} = a_{(i+1)j}$. For example:
The output \((D_{\text{flat}})\) of the tabular transformation phase.

3.2 Derived Operations

A derived operation is an operation that can be achieved by using a sequence of primitive operations. For example, by using a \texttt{Select} operation with row indices (or column indices) in the reverse order, we can create a new operation, \texttt{Flip}, which can flip a table vertically (or horizontally). Though there are only five primitive operations, executing primitive operations with different orders and options can produce many different effects. The ability that new operations can be derived from existing ones makes the number and power of table operations extendable and, thus, allows virtually unlimited layouts (in the first normal form).

There are many interesting derived operations. However, for brevity, we will discuss only \texttt{RotateRight} and \texttt{Resize} here. A \texttt{RotateRight} operation right rotates the source table by 90 degrees. The \texttt{RotateRight} operation can be achieved by a sequence of \texttt{Contract}, \texttt{Expand}, and \texttt{Flip} operations. A \texttt{Resize} operation changes either the width (number of columns) or the height (number of rows) of a source table. It can be achieved by using two consecutive \texttt{Contract} operations followed by two consecutive \texttt{Expand} operations.

To facilitate report layout creation, in addition to the 5 primitive table operations described in Sect. 3.1, TPS provides 15 built-in derived table operations that offer the most frequently used transformation operations, including \texttt{Resize}, \texttt{Transpose}, \texttt{HorizontalFlip}, \texttt{RotateRight}, \texttt{MergeRight}, \texttt{Put} (put a table to the right side of another table), \texttt{MergeTop} (put a table on top of another table), etc.

3.3 Example

This section uses the report shown in Fig. 1 as an example to explain how to use table operations to transfer source tables (Fig. 2) into a flat table \(R\) (Fig. 6) that shows the reservation arrangement and prices. The layout of \(R\) is similar to the subtables shown in Fig. 1, except that \(R\) has an extra column that is used as the key to further separate \(R\) into subtables in the next phase. \(R\) has three areas \(R_A\), \(R_B\), and \(R_C\) (indicated by dash borders). \(R_A\) shows the name of the show and the row number. \(R_C\) shows the price of each row. Both \(R_A\) and \(R_C\) can be obtained from \texttt{Prices} (Fig. 2). \(R_B\) shows the reservations in 2D format and can be obtained by using a \texttt{Match} operation. Thus, \(R\) can be produced by the following steps:

1. Use a \texttt{Select} operation (Fig. 7-a) to produce \(R_A\).
2. Use a \texttt{Select} operation (Fig. 7-b) to produce \(R_C\).
3. Use a \texttt{Match} operation (Fig. 7-c) to produce \(R_B\).
4. Merge \(R_A\), \(R_B\), and \(R_C\) to produce \(R\) (Fig. 6).

Steps 1 and 2 are simple, but step 3 is a little more complicated. To produce \(R_B\), the \texttt{Match} operation used in step 3 needs two arguments (tables): \texttt{Map} and \texttt{Data} (Fig. 8). These two tables can be obtained by the following substeps:

3-1 Define \texttt{MapH} as shown in Fig. 9-a.
3-2 Produce \texttt{MapV} as shown in Fig. 9-b.
3-3 Merge \texttt{MapH} and \texttt{MapV} to produce \texttt{Map}.
3-4 Produce \texttt{DataA} as shown in Fig. 10-a.
3-5 Produce \texttt{DataB} as shown in Fig. 10-b.
3-6 Merge $D_{A}$ and $D_{B}$ to produce $D_{A}$.

Steps 3-1 to 3-3 generate $M_{A}^{P}$ by producing and merging $M_{A}^{P}$ and $M_{V}^{P}$. $M_{A}^{P}$ represents the numbering and ordering of seats, which is stationary and can be defined as a constant table (Fig. 9-a). $M_{V}^{P}$ is the vertical key and can be obtained from $R_{A}$. Steps 3-4 to 3-6 produce $D_{A}$ from $R_{A}$ by contracting the first two columns ($D_{A}$) and merging the result with the last two columns ($D_{A}$). $R_{B}$ (from steps 3-1 to 3-6) is now ready. Step 4 uses two merge operations to merge $R_{A}$, $R_{B}$, and $R_{C}$ to produce $R$. Note that $R$ is a flat table. Section 4 will discuss how to restructure flat tables into nested ones. For brevity, the visual programming user interface that accomplishes the above steps is omitted. Readers who are interested in the user interface may refer to [25] for more information.

### 4. Nested Structure

The goal of the third phase, nested structure, is to transfer flat tables ($D_{A}$) into a single nested table ($D_{nested}$) to represent the nested report. For example, the flat table $R$ shown in Fig. 6 is transferred in this phase into a nested table shown in Fig. 11. A nested table contains a number of subtables obtained from $D_{A}$. To specify the relationship (including both parent-child relationship and display order) of the subtables, a report structure tree (called simply report tree) is proposed. As an example, Fig. 12-a is a nested table with 4 subtables that are obtained from $D_{A}$. The structure of Fig. 12-a can be specified by the report tree shown in Fig. 12-b. A root node (root) presents the report. The root contains a container node (container) that contains 4 table nodes.

<table>
<thead>
<tr>
<th>Node name</th>
<th>Icon</th>
<th>Functionality</th>
<th>MBCO</th>
<th>Leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td>📊</td>
<td>Root of report tree</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>🌟</td>
<td>An arbitrary text</td>
<td>🇱🇷 or 🇬🇧</td>
<td>Yes</td>
</tr>
<tr>
<td>Table</td>
<td>⬜️</td>
<td>A (imported) table</td>
<td>🇱🇷 or 🇬🇧</td>
<td>No</td>
</tr>
<tr>
<td>Container</td>
<td>⬜️</td>
<td>A subtree (a nested sub-report)</td>
<td>🇱🇷 or 🇬🇧</td>
<td>No</td>
</tr>
<tr>
<td>Schema</td>
<td>⬜️</td>
<td>A column of table</td>
<td>🌟</td>
<td>Yes</td>
</tr>
<tr>
<td>Vertical Aggregation</td>
<td>⬜️</td>
<td>Perform a horizontal aggregation</td>
<td>🌟</td>
<td>Yes</td>
</tr>
<tr>
<td>Vertical Operation</td>
<td>⬜️</td>
<td>Provide vertical operations</td>
<td>🌟 or ⬜️</td>
<td>No</td>
</tr>
<tr>
<td>Vertical Aggregation</td>
<td>⬜️</td>
<td>Perform one or more vertical aggregations</td>
<td>🌟</td>
<td>Yes</td>
</tr>
<tr>
<td>Caption</td>
<td>⬜️</td>
<td>Insert a caption row</td>
<td>🌟</td>
<td>Yes</td>
</tr>
<tr>
<td>Separator</td>
<td>⬜️</td>
<td>Insert a separator between rows</td>
<td>🌟</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The “MBCO (Must be a child of)” column specifies the syntax requirement of report trees. The nodes that must be a child of Report node are called basic nodes (i.e., Text, Table, and

...
Container), and the others are called child nodes. For brevity, the grammar of the report tree is omitted. Please refer to [27] for an EBNF grammar. The following sub-sections describe the syntax and semantic of tree nodes.

4.1 Basic Nodes

The node Report (） is the root of a report tree. It may contain three kinds of nodes, Text node, Table node, and Container node, called basic nodes. There are no constraints on the number and ordering of basic nodes when they are attached under a Report.

A Text node (） is a leaf node presenting an arbitrary text. A Table node represents a table that belongs to \( D_{flat} \). Unlike Text node, a Table node is not a leaf node. The appearance of a Table node (） can be modified or enhanced by adding child nodes discussed in the next section.

A Container node (） is the root of a subtree, which groups a set of basic nodes into a logical unit. A Container node defines the layout of its children by its Layout and Size properties. There are two types of layouts: Zigzag (arrange children from left to right and restart at a new row when the specified size boundary is reached) and Reverse-N (arrange children from top to bottom and restart at a new column when the specified size boundary is reached). A Container node may contain another Container node. Thus, tables in \( D_{flat} \) can be arbitrarily nested. Figure 13-a gives an example that Container A contains Container B. Note that because of the Layout property, the children of these two nodes are displayed in Zigzag and Reverse-N order, respectively.

4.2 Child Nodes

A Table node may contain three kinds of child nodes: Schema node, Horizontal Aggregation node, and Vertical Operation node. They are used to modify the behaviors of their parent table node. A Schema node (） represents a column of data. It is generated automatically when its parent Table node becomes a node in the tree. The properties of a Schema node can be used to show/hide the title of a column (Show Title), show/hide the entire column (Show Contents), and change the ordering of columns (Index).

A Horizontal Aggregation node (） performs an aggregation to the cells of a specified row, and creates a new derived column. A Table node may contain several Horizontal Aggregation nodes. Note that, sometimes, the same effect can also be achieved by using SQL statements during data query phase. However, SQL statements cannot be used to perform aggregations for \( D_{flat} \) (see Fig. 3).

Vertical operations are also offered. To avoid confusion with horizontal operations (schemas and aggregations), all vertical operations (e.g., Vertical Aggregation nodes) are collected under a Vertical Operation node (）。 A Table node may contain (optionally) one and only one Vertical Operation node.

A table can be divided into a list of subtables by turning on Group property of Vertical Operation node. The rows with the same values in the Key Column will be grouped together and treated as a subtable, resulting a list of subtables. An example is shown in Fig. 14, where T1 and T2 are used as Key Columns (Fig. 14-a), and Table 1 (Fig. 14-b) is turned into a list of subtables (Fig. 14-c). Each subtable is considered as a regular table and its layout follows the rules defined by its parent container (i.e., Zigzag with Size 2 in this case). It is possible to further divide subtables into sub-subtables by attaching a Vertical Operation node under another Vertical Operation node. There is no limitation to the level of nesting.

A Vertical Operation node may optionally contain three kinds of child nodes: Vertical Aggregation node, Caption node, and Separator node. Aggregation of table cells in the same column is achieved by using Vertical Aggregation node (）， which creates a new derived row. A Vertical Aggregation node can also perform aggregation for several columns simultaneously.

A Caption node (） adds a caption for the table. The type, position, and alignment of a caption are defined by its properties. A Separator node (） adds a line to separate rows.

4.3 Example

This section demonstrates the steps needed to transfer Fig. 6 to Fig. 11. Note that Fig. 11 is almost the same as the final report (Fig. 1) except that it is not formatted. The most prominent difference between Fig. 6 and Fig. 11 is that each show is presented as a single subtable in Fig. 11. The following six steps constructs a report tree (Fig. 15) to achieve the transformation:

1. Add a Container node as a child of the root node (Report). The Container node is used to specify the layout (to be done in step 4) of the underlying subtables.
2. Add \( R \) as a child Table node of Container node. Since \( R \) has seven columns, seven Schema nodes will be generated automatically.

3. Add a Vertical Operation node as a child node of \( R \). Turn on its Group property and specify Key Column property as 1, indicating that the first column (show name) is used as the key to further divide \( R \) into subtables.

4. Specify Layout (i.e., Zigzag) and Size (i.e., 2) properties of Container node.

5. Hide unnecessary contents by turning off Show Contents and Show Title properties for Column 1 node, and Show Title property for Column 2 node.

6. Add two Caption nodes as the children of Vertical Operation node and specify their properties. The Add Type property (i.e., Above) specifies that the caption is placed above the subtables. The node Show Caption shows the caption for each subtable. Note that “Content:15” indicates that the content of the caption is the value of the first column (i.e., show name).

So far, we have transferred source tables (Fig. 2) into an unformatted nested report (Fig. 11). For brevity, the user interface that accomplishes the above steps is omitted. Please refer to [26] for more information. Note that Fig. 11 is called a two-level nested table, because it is a table (dash-bordered) that contains subtables. The next section will describe the methodology of formatting multi-level nested tables.

5. Report Format

The goal of the fourth phase, report format, is to specify formatting attributes (e.g., fonts and colors) for the nested report (\( D_{nested} \)) to make it more readable. We propose a WYSIWYG format editor and a set of formatting rules designed specifically for nested reports. It is possible to specify formatting attributes by enhancing the nodes of the report tree to include formatting properties. However, such a method is not WYSIWYG — the report designer cannot directly use the final report to specify formatting attributes (e.g., select a piece of text in the report and change its font). Most report tools [24] have the same problem. They offer report designers a template editor to specify the format of the template. In this case, the formatting attributes are applied to the template, not directly to the final report. Thus, a template editor is not as friendly as a real WYSIWYG format editor.

The proposed format editor employs “programming by example” methodology [4], which is also known as programming by demonstration or demonstrational programming. Programming by example is an end-user development technique in which the developer performs actions on concrete examples until the desired outputs are created. The system records user actions and infers a generalized program or script that can be used upon new examples. To an end-user developer, viewing the outputs of concrete examples is in general a lot easier than deriving a program (or rule). Therefore, our formatting editor allows an end-user developer to perform formatting without learning the syntax and semantics of formatting rules.

The report designer uses the editor to directly view the final report (with example data) and specify report formats. The format editor is aware of the nested structure of the report. For example, there is an operation that allows the report designer to click on the Price column of an arbitrary table to select all Price columns of all tables. Therefore, the report designer does not need to perform repetitious operations. When the report designer specifies formats, his/her operations are translated into a sequence of formatting rules (e.g., “set the font style of the Price column bold”; see Fig. 16). The rules are independent of the example data. When a report viewer requests the report (Fig. 4), the report is reconstructed and formatted according to the rules.

5.1 Formatting Rules

There are two kinds of formatting rules. One specifies the properties of the entire report, called document property. The other specifies the formatting attribute (e.g., font size) for a selected formatting region (e.g., the Price columns), called regional rule. A formatting rule is denoted as either Rule (Document-Property) or Rule (Region, Attribute). The usage of formatting rules will be explained in the following sub-sections.
### 5.1.1 Document Properties

A document property sets a particular attribute for the report. Most document properties are designed for specific display formats. For example, if the display format is HTML, report designers can set header title, language, and link colors properties. On the other hand, if the display format is PDF, report designers can set author, page size, page margins, and document title properties.

The page break of a nested report is specified by the paging rule property. For HTML, the default paging rule is “not to page.” For PDF, the default paging rule is “fit into the page size.” Sometimes, it is desirable to set page breaks based on table structures. For example, suppose that a two-level Sales report is grouped by the department, it is natural to display one department in one page. In this case, report designers can set the paging rule to add a page break for each department (i.e., level 1). For brevity, the discussion for the rest of the document properties is omitted.

### 5.1.2 Regional Rules

A regional rule specifies both a formatting region that selects an area of the report, and a formatting attribute that modifies the appearance of the selected area. A formatting region is specified by indicating its level and scope, where level is a non-negative integer and scope may be one of the following five values: table, row, column, cell, and content. For example, given the three-level nested report shown in Fig. 17-a, the rule “Region (level: 2, scope: table) all” selects all (sub-)tables at level 2 (Fig. 17-b). A scope may be refined by using parameters given in Table 2. The all (Yes/No) parameter (default Yes, if unspecified) indicates whether all items of the same scope are selected. If all is No, the parameter number (integer) is used to specify that the number-th item of the same scope are selected. For example, the rule “Region (level: 2, scope: table, all: No, number: 3)” selects only the 3rd table at level 2 (Fig. 17-c).

In case that the scope is row, table rows are selected by specifying their positions in the tables. For example, the area selected by “Region (level: 2, scope: row, all: Yes, base: top-down, offset: 0)” is “the first rows (counting from top to bottom) of all subtables at level 2,” i.e. the dash-bordered areas shown in Fig. 17-d. Another example: the area selected by “Region (level: 2, scope: row, all: No, number: 6, base: top-down, offset: 0)” is “the first row (counting from top to bottom) of the 6th subtable at level 2,” i.e., the dash-bordered area shown in Fig. 17-e. In contrast, if level is 1, the area selected by Region (level: 1, scope: row, all: Yes, base: top-down, offset: 0) is shown in Fig. 17-f. The rest of the scopes, column, cell, and content, are used in a similar way, except that they select columns, cells, contents of cells, respectively.

After specifying the formatting region, a formatting attribute (Table 3) can be applied. A formatting attribute is scope-sensitive. For example, the font attribute can only be applied to content (text). We use “Attribute (Name: Value)” to denote a formatting attribute. For example, the formatting attribute “attribute (background-color: gray)” sets the background color as gray. We are now ready

![Fig. 17 A three-level nested tables and different formatting regions.](image)
Table 3  The supported formatting attributes.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>table, row, column, cell</td>
<td>background-color</td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td>border</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>border-color</td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td>height</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>width</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>alignment</td>
<td>left/right/center</td>
</tr>
<tr>
<td>content</td>
<td>color</td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td>font</td>
<td>Font</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>style</td>
<td>bold/italic/underline</td>
</tr>
<tr>
<td></td>
<td>anchor</td>
<td>String</td>
</tr>
</tbody>
</table>

Table 4  The rules that format the final report.

<table>
<thead>
<tr>
<th>Purpose / Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Set the background to gray for Price columns of all level-1 tables.</td>
</tr>
<tr>
<td>Rule [Region [level: 1, scope: column, base: right-to-left, offset: 0], Attribute [background-color: gray]]</td>
</tr>
<tr>
<td>2. Align all level-1 texts to center.</td>
</tr>
<tr>
<td>Rule [Region [level: 1, scope: table], Attribute [alignment: center]]</td>
</tr>
<tr>
<td>3. Remove the border of the first rows of all level-1 tables.</td>
</tr>
<tr>
<td>Rule [Region [level: 1, scope: row, base: top-down, offset: 0], Attribute [border: 0]]</td>
</tr>
<tr>
<td>4. Align the caption “SHOWS” to center.</td>
</tr>
<tr>
<td>Rule [Region [level: 0, scope: row, all: No, number: 0, base: top-down, offset: 0], Attribute [alignment: center]]</td>
</tr>
<tr>
<td>5. Remove the background of the Price cells of all level-1 tables.</td>
</tr>
<tr>
<td>Rule [Region [level: 1, scope: cell, base: top-right, row-offset: 0, column-offset: 0], Attribute [background-color: none]]</td>
</tr>
<tr>
<td>6. Set the color of the captions of all shows to blue.</td>
</tr>
<tr>
<td>Rule [Region [level: 1, scope: row, base: top-down, offset: 0], Attribute [color: blue]]</td>
</tr>
<tr>
<td>7. Set page size.</td>
</tr>
<tr>
<td>Rule [page-size: A4]</td>
</tr>
<tr>
<td>8. Set page margins.</td>
</tr>
<tr>
<td>Rule [top-page-margin: 3.17cm]</td>
</tr>
<tr>
<td>9. Set paging rule.</td>
</tr>
<tr>
<td>Rule [paging-rule: level-1]</td>
</tr>
</tbody>
</table>

We now use Fig. 11 as an example to demonstrate how to format it into the final report (Fig. 1) in PDF format. Table 4 lists all the steps and rules that are used. Note that the rules are generated automatically by the format editor whenever an area is selected and formatted. The first step sets the background to gray for the Price columns of all level-1 tables. The report designer chooses level 1 from the combo box “Level...” sets column scope by clicking “...”, and checks all parameter “...”. All Price columns of level-1 tables will be highlighted (Fig. 19). Then, the report designer clicks the button “...” (area B) to set the background color to gray. The results are shown immediately in area C. The rest of the steps are manipulated in a similar way, and their explanations are omitted for brevity.

5.2 Example

The user interface of the proposed WYSIWYG format editor is shown in Fig. 18. There are five areas (marked by heavy lines). The Import button (in area E) is used to import the nested report ($D_{nested}$) into area C. When the report is imported, the report designer may choose the desired level and scope, and specifies additional parameters if necessary (from area A). Then, the report designer clicks the data displayed in area C. The GUI will highlight the selected area (e.g., Fig. 19). The selected area can be refined by changing either level, scope, or related parameters. When the report designer is satisfied with the selection, the toolbar in area B can be used to apply formatting attributes to the selected area, and the report shown in area C will immediately reflect the change, giving report designers the feel of using a WYSIWYG format editor. The document type (either PDF or HTML) and its properties are specified in area E.

Fig. 19  The highlighted selected area.

Fig. 18  The formatting interface.

to present a complete regional rule — the regional rule that “sets background color as gray for all Price columns of all tables” is specified by “Rule [Region [level: 1, scope: column, base: right-to-left, offset: 0], Attribute [background-color: gray]].” Note that it is possible to include many more formatting attributes into Table 3. We did not do so, because our goal is to show that the proposed approach works, rather than creating a full commercial-grade report tool.

6. Related Work

Most commercial[5]–[8] and open-source[9] tools use
schema-based approach for report generation. Oracle Report Builder [5] provides a number of predefined templates for frequently-used reports, and wizards to help query and map data into a template. The other tools [6]–[9] offer a set of predefined template components for the report designer to create a report template. Each template component can be configured to accomplish some layout transformations. Unfortunately, there are limitations to the use of template components. In particular, not all template components can cooperate with the other components, limiting the varieties of report layouts that can be produced. In comparison, the table operations proposed in this paper can be arbitrarily cascaded. Thus, table operations are more flexible than template components.

There are researches studying the generation of reports. In particular, Tarassenko [3] proposed a reporter system that can produce sophisticated read-to-use reports. By using the reporter system, the development of a report is focusing at describing tables with column/rows, selecting and summarizing data, and formatting in the destination document. A document-driven approach to report generation is proposed by Chan [2]. Contents of a report can be specified by using a transformation language together with queries that retrieve data from different databases. However, both of them allow only simple and regular transformations, limiting the kinds of reports that can be produced.

Masuishi [4] developed a reporting tool based on the programming by example concept to simplify layout transformation. The tool enables users to design a sample layout of an example row of relational table data, and select an iteration pattern for the sample layout. Although it is more flexible than the traditional schema-driven approach, nested reports are not supported.

Since the advent of Web 2.0, Web applications with dynamic web pages consisting of database reports are all over the Internet. Script languages are commonly used to generate web-page reports. For example, PHP [10] can be used to generate database reports easily. However, using PHP or other languages [11]–[13] to produce reports is a low-level programming task, and is not suitable for a report designer without programming expertise. In contrast, the methodology proposed in this paper offers high-level abstractions for report generation and allows end-user programming.

Over the years, a rich literature that uses non-first-normal-form (NF2) relations as data models for database applications has been developed [19]. These results are extendable to nested report generation. However, it is somewhat more difficult for an end user to understand NF2 relations. In this paper, the effect of nested tables are produced by using report trees, which are more friendly to end users.

Gyssens et al. [17] proposed a tabular database model and a tabular algebra (TA) as the language to query and restructure tabular data. Many restructuring operations of TA are useful for report generation. These operations are either achieved by table operations (e.g., the effect of a TA GROUP operation is similar to a two-dimensional FETCH table operation) or the report structure tree nodes (e.g., the effect of a TA SPLIT operation is similar to turning on the Group property of Vertical Operation node).

Most current report tools [6]–[9] allow report designers to specify report format on the template of the report being produced. Tough it works, it is a step short of being WYSIWYG, which can be more friendly and intuitive to end-users [23]. In contrast, this paper proposes a WYSIWYG format editor that is aware of the structure of the nested report to enable WYSIWYG formatting operations. Wang [20] proposed a layout specification to handle the formatting of a table. The specification contains two parts. The first one, called topological specification, decides the position of table content. The other one, style specification, specifies the formatting attributes on the table. Silverhorn [21] extended the specification [20] and developed a tool, called TabularMagica. However, both of the two approaches do not support nested reports.

7. Evaluation

Can end-user developers use TPS to create reports? We conducted three experiments to evaluate the proposed tabular transformation, nested structure, and report format phases. Each experiment invited a number of participants who were given a short lecture with a step-by-step example covering the concept and manipulation of the user interface. Then, the participants were requested to apply the principles and skills learned from the lecture to accomplish an assigned task, which was different from the example and used different operations (table operations, nodes, or formatting rules) in different orderings and with different parameters.

For the tabular transformation phase, the participants were 34 freshmen students invited from the Electrical Engineering and Computer Science department of National Taipei University of Technology (NTUT). After a 25-minute lecture on table operations, the participants were given 3 source tables as $D_{source}$ and requested to transform $D_{source}$ into a single table ($D_{flat}$) in 50 minutes. The transformation required a total of 12 table operations and was, in fact, more complicated than the example shown in Sect. 3.3. At the end of the experiment, all (100%) participants were able to accomplish the task. On average, 16 minutes were used.

For the nested structure phase, the participants were 45 freshman students invited from the Computer Science and Information Engineering department of NTUT. After a 20-minute lecture on report trees, the participants were given a predefined table (similar to Fig. 6) as $D_{flat}$ and requested to create a report tree that converted $D_{flat}$ into a report structure $D_{nested}$ (similar to Fig. 1) in 30 minutes. At the end of the experiment, 44 out of 45 participants (97.7%) were able to accomplish the task. On average, the 44 successful participants used 12 minutes.

For the report format phase, the same students participated in the nested structure phase were requested to perform an additional experiment. After a 15-minute lecture on the WYSIWYG format editor, the participants were given a predefined $D_{nested}$ similar to Fig. 1 and requested to pro-
duce a complicated nested report format (using a total of 22 formatting rules) in 20 minutes. Again, 44 out of 45 participants (97.7%) were able to accomplish the task. On average, the 44 successful participants used 6 minutes.

Overall, the experiment results indicated that a very high percentage (over 97.7%) of participants were able to quickly grasp the concepts of table operations, report trees, and nested-report formatting. At the end of each experiment, a few questions were used to survey the participants. For each question, the participants can specify their level of agreement in 5-point Likert scale [22]: strongly agree (5), agree (4), neutral (3), disagree (2), or strongly disagree (1). The survey question “is WYSIWYG format editor easy to learn?” and “is WYSIWYG format editor straightforward to use?” received an average score 4.1 and 4.0, respectively, rejecting the null hypothesis (the participants are neutral to the question) at 95% confidence level. The survey results of table operations (scored 3.7 and 3.6) and report trees (scored 3.4 and 3.4) were similar, also rejecting null hypothesis. Therefore, we can conclude that the participants considered the concepts of these three phases easy to learn and straightforward to use.

The participants of the experiments were freshmen. Although, at the time of the experiment, the participants had already finished 3 credits of programming course, none of the participants had any experience with either database or report generation. Therefore, the experiment results are valid for end-users with engineering background. End-users with other backgrounds (e.g., art and literature) will need another evaluation, which is beyond the scope of this paper.

8. Comparison

A limitation of the proposed approach is that reports must be presented in tabular formats. This limitation is almost negligible, since a non-tabular report can also be considered as a table without any borders. To demonstrate the strength of the proposed approach, we will compare TPS with a commercial report tool, showing that TPS can produce almost all reports supported by the commercial report tool, but not the other way around.

We chose Oracle Report Builder (ORB) [5] as the comparison target, because ORB is a well-known and powerful commercial report tool. The other report tools [6]–[9] support similar, though not exactly the same, functionalities. ORB supports 22 different report templates (the second column of Table 5). We use TPS to imitate what ORB can do. The results are shown in Table 5. TPS can produce almost all reports (2075 ≈ 90%) supported by ORB. But, what about the other way around? Can ORB imitate what TPS can do? The answer is no. Without writing additional programs, ORB cannot produce any reports with effects such as resize, rotate, transpose etc. In fact, we can name an unlimited number of TPS reports that cannot be supported by ORB. For example, ORB cannot produce useful reports, such as Fig. 1 in this paper, Fig. 1 in [4], Table 2.3 in [20], Fig. 2 and 8 in [25], and Fig. 1 in [26]. Therefore, we can conclude that TPS is more flexible than ORB (in terms of report layout capability).

Table 6 gives a comparison of different report generation methodologies, where nesting indicates whether a methodology supports nested reports, flexibility indicates whether a methodology is flexible in producing different kinds of reports, and difficulty indicates whether a methodology is easy-to-learn and easy-to-use to an end-user report designer. Writing report generation programs (e.g., [10]–[13]) is the most flexible but also the most difficult methodology. Both extended query [16], [17] and NF2 relational models [18], [19] can support nested reports and are highly flexible. However, they are also difficult to end-users. Researches [2]–[4], tabular models [20], [21], and visual query [14], [15] are easier to use, but they do not support nested reports. Report tools [5]–[9] support nested reports and are easy to use, but they are not very flexible. Overall speaking, among the methodologies that support nested reports, TPS is the only one that is both easy to use and highly flexible.

9. Conclusion

This paper proposed a report generation approach that integrates several techniques including table operations, report
trees, and nested formatting rules. A visual report tool with a friendly GUI, called TPS, was implemented to demonstrate the use of the proposed approach. TPS is capable of producing all sorts of nested reports with sophisticated formats, and is more flexible than other approaches such as [2]–[9], [14]–[21]. With TPS, writing dedicated programs to transfer report layouts and specify report formats is no longer necessary. Our experience with TPS indicated that it is simple and powerful — nested reports with sophisticated layouts and formats can be generated without difficulty. So far, the proposed approach does not support form-based interactions, which would be useful if the report is to be accessed by a browser. In the future, we expect to extend the proposed approach to support such interactions.

Acknowledgements

This research was partially supported by the National Science Council and Ministry of Economic Affair, Taiwan, under contract number NSC 99-2220-E-027-005 and MOEA 100-EC-17-A-02-S1-135, which are gratefully acknowledged. This research was partially supported by the National Science Council and Ministry of Economic Affairs, Taiwan, under contract number NSC 99-2220-E-027-005. We also thank anonymous reviewers who provided many useful suggestions for improving this paper.

References